Homework Module:

Evolving Neural Networks for a Minimally-Cognitive Agent

Purpose: Learn to implement an evolving neural network to use as a controller for a simulated agent that performs a task requiring a small amount of intelligence.

1 Assignment

You will use your evolutionary algorithm (EA) to evolve the weights and and other parameters for a small artificial neural network (ANN) whose fitness will depend upon the performance of an agent controlled by the ANN. This agent will do a *minimally cognitive* task similar to those used in the field of artificial life (ALife) to illustrate the emergence of rudimentary intelligence. Specifically, the agent will sense falling objects of various sizes and must evolve the ability to a) intercept objects smaller than itself, but b) avoid objects of its own size or larger.

2 Minimally-Cognitive Tasks

Beginning in the early 1990's, Randall Beer (then at Case Western Reserve University and now at Indiana University) and his graduate students produced a series of research papers[4, 1, 2, 3] on the evolution of ANNs for controlling simulated agents that perform tasks reminiscent of 1970's video games. In most cases, the agent is a simple ball that can only move horizontally along the bottom of the screen, while objects of various sizes and shapes fall from the sky. The agent has several primitive sensors that enable it to detect portions of the falling objects. The agent's ANN then integrates these sensory inputs to determine a course of action: stay put, move left or move right, with varying velocities. When the object reaches the bottom of the screen, the agent should either *catch* it, essentially by just *being there* at the point where the object hits bottom, or avoid it, by distancing itself from the landing spot.

Figure 1 depicts one of the more popular scenarios in which falling objects are either circles or diamonds. The agent should catch the circles but avoid the diamonds. Other scenarios involve combinations of falling objects and often require more sophisticated cognition, such as the ability to compare the sizes of two falling objects.

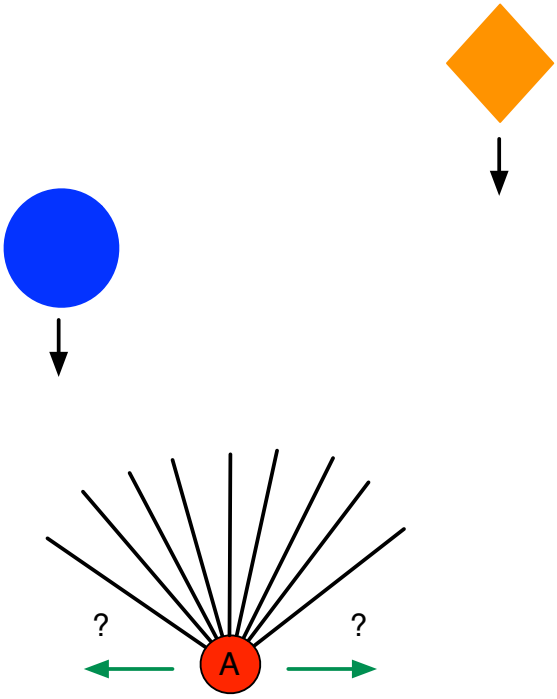

Figure 1: Randall Beer's classic video-game agent (A) with multiple *visual* sensors (depicted as lines emanating from the agent) and the capability of horizontal movement.

3 Continuous Time Recurrent Neural Networks

Via his minimally-cognitive behavior research, Randall Beer has popularized an interesting new style of ANN: Continuous-Time Recurrent Neural Networks (CTRNNs). These supplement standard ANNs with two neural properties: a time constant and a gain. Values of both properties normally vary among neurons. Beer often describes the CTRNN neural model in a dynamic-systems format, while the following reformulation adheres more closely to the standard integrate-and-fire perspective.

First, integrating the inputs from all upstream neighbors involves the standard equation (1):

$$s_i = \sum_{j=1}^{n} o_j w_{i,j} + I_i \tag{1}$$

Here, o_j is the output (or, equivalently, the activation level) of neuron j, while $w_{i,j}$ is the weight on the arc from neuron j to neuron i. I_i is the sum of all external inputs to neuron i. A typical external input is the value of the single sensor associated with an input neuron. Non-input neurons typically have no external inputs.

In Beer's CTRNN descriptions, y_i denotes the internal state of neuron i. Equation 2 shows that the derivative of this internal state, $\frac{dy_i}{dt}$, is a combination of s_i and a *leak term*, wherein a portion $(\frac{y_i}{\tau_i})$ of the internal state from the previous time step drains out. In addition, Beer incorporates a neuron-specific *bias* term, θ_i. As shown later, this is easily modeled as an input from a *bias neuron* and is thus incorporated into the s_i term.

The time constant, τ_i, determines how fast the neuron changes internal state. A low value entails fast change, with the new state being predominantly determined by current conditions; whereas a high τ_i produces more *memory* in the neuron for its previous state(s) (due to less leak and less influence of s_i), which are reflections of earlier conditions. It is this memory, and the ability of neurons to vary in their levels of it (via different τ values) that enables CTRNNs to exhibit extremely rich dynamics, and hence, convincingly sophisticated cognition. This assignment only scratches the surface of those cognitive abilities.

$$\frac{dy_i}{dt} = \frac{1}{\tau_i}[-y_i + s_i + \theta_i] \tag{2}$$

As shown in equation 3, Beer typically employs a logistic (i.e. sigmoidal) activation function to convert the internal state to an output, o_i. However, y_i is multiplied by the neuron-specific gain term, g_i.

$$o_i = \frac{1}{1 + e^{-g_i y_i}} \tag{3}$$

On each timestep, a CTRNN neuron must recompute s_i and then $\frac{dy_i}{dt}$, from which new values of y_i and o_i are then calculated. The new value of y_i is simply the old value plus $\frac{dy_i}{dt}$, while o_i stems from equation 3.

The CTRNN model is used in all of Beer's minimally-cognitive simulations, spread across dozens of interesting research papers; but the ranges of key parameters (gains, biases, weights and time constants) vary according to the demands of the different minimally-cognitive tasks. The network topologies typically consist of an input layer - with each neuron attached to one sensor and simply outputting the value of that sensor - a hidden layer, and an output (motor) layer, which controls the left and right movement of the agent. The *recurrence* in CTRNNs normally stems from a) connections among the neurons of the hidden layer, b) backward links from the motor neurons to the hidden layer, c) connections between the two motor neurons, and/or d) connections from individual hidden (or motor) neurons to themselves. Recurrence adds a second form of memory to the CTRNNs, since the states of neurons are fed back into the system.

4 The Tracker

Though simple, even the video games of the 1970's (i.e. your professor's childhood) do involve small amounts of calculation to implement. For example, in the agent model of Figure 1, each of the visual sensors requires geometric calculations to determine whether or not (and at what point) each sensor's ray intersects the falling object. In an attempt to *cut to the chase* and avoid a lot of this (relatively unnecessary work), we can further simplify these video-game scenarios.

We can combine a discrete abstraction of continuous space along with a simpler form of sensor to achieve an even more manageable task scenario, but one which still requires minimally-cognitive behavior of the agent, which we will call a *tracker*. As shown in Figure 2, the tracker agent now consists of a row of *shadow sensors*; these detect portions of the falling object that are directly overhead, *and at any vertical distance from the bottom of the screen*. Objects consist of a fixed number of squares, each of which fills one grid cell in the discrete environment, as shown in the figure. Objects fall at a speed of one vertical level per timestep, and they may fall straight down or with a horizontal velocity of one column (to the left or right) per timestep. The tracker's task is to intercept objects narrower than itself and to avoid contact with all other objects.

Detecting shade-sensor activation then becomes a trivial comparison of the horizontal positions of the object's squares and the agent's sensors, as shown by the dashed vertical lines (representing shadows) in the figure.

The whole scenario is easily represented as a binary array, with two chunks of 1's, denoting the agent and the falling object, both of which may move with each timestep. Alternatively, the agent and object can be modeled as having a width and a cartesian location. Comparisons of these width-location pairs are easily translated into sensor activations. These representations are denoted Rep-1 and Rep-2, respectively.

As one more simplification, it is wise to include *wrap-around* in the environment: if the object or agent tries to move past the leftmost (rightmost) edge, it will reappear coming in from the right (left) side. Though this adds a little complexity to the calculations, especially when using Rep-2, it makes the task a bit easier for the evolving CTRNN to solve.

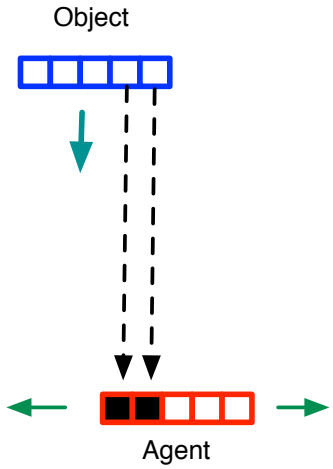

Figure 2: A simplified version of Randall Beer's classic video-game scenario. Here, objects cast a *shadow* straight down (dashed vertical arrows), and the tracker agent consists of a row of *shadow sensors* (red squares). The agent can only move horizontally, while the objects fall straight down. The agent captures the object if the overlap between the agent's sensor group and the object is above a user-defined threshold (e.g. 80 %) when the object reaches the bottom.

4.1 Details of the Tracker Environment

You are free to experiment with any number of Tracker environments. What follows are parameters that are known to give reasonable results with the evolving CTRNN described below.

For this exercise, the agent should be 5 cells wide: it should have 5 shadow sensors. The falling objects should have sizes between 1 and 6. The entire arena should be 30 units wide and 15 units high. A *capture* should be defined as a situation where each section of the falling object lands on a shadow sensor, while *avoidance* occurs when **none** of those sections land on a sensor. In short, avoidance is more challenging than just failing to capture.

The agent should be able to move at most 4 horizontal units per timestep, in either direction. You will need to figure out how to use the outputs of the two motor neurons to determine the direction and magnitude of each move. It is advisable to include wrap-around in your environment, since it can be hard for the controller to learn to move the other direction when it hits a wall.

You are strongly advised to build a simple graphic display to show the performance of your agents. This will make debugging easier, and the project more fun.

5 Evolving the CTRNN for a Tracker Agent

Figure 3 portrays a CTRNN topology that suffices to solve simple tracker problems. The 5 input neurons correspond to the 5 shadow sensors of the agent. These simply output a 1 or 0, depending upon whether or not a portion of the falling object is directly above that sensor. The hidden and output/motor layers each consist of 2 neurons, all of which abide by the CTRNN integrate-and-fire rules of equations 1, 2, and 3. The input layer is fully connected to the hidden layer, which is fully connected to the output layer. Recurrence stems from intra-connections in the hidden and output layers.

To account for the bias terms, θ_i in equation 2, a standard technique is to include a single bias node (labeled B in Figure 3. This node always outputs a 1, but the weights on the connections from B to each of the hidden and output nodes can vary. In this case, they are evolved. Hence, evolution determines the biasing input to each node.

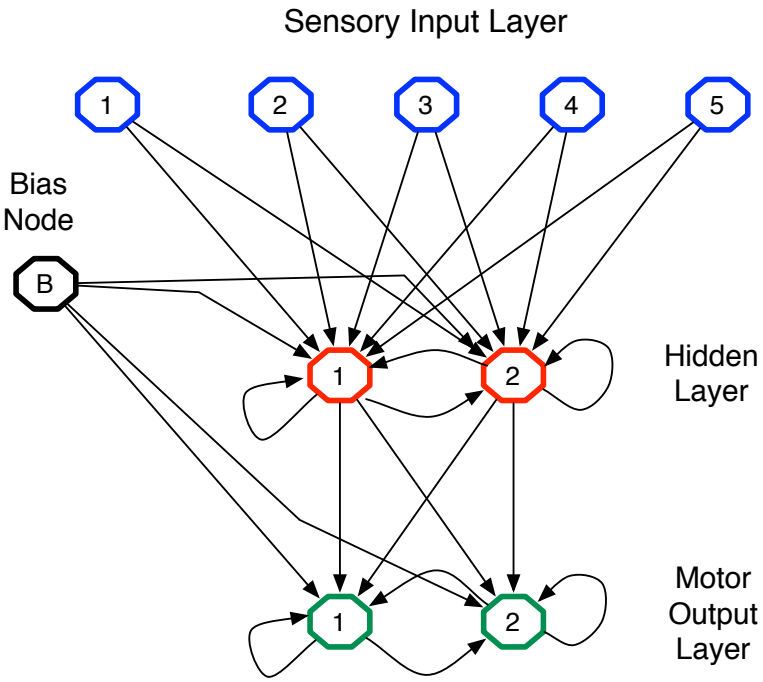

Figure 3: A CTRNN topology for evolving a tracker. All connections are shown (thin arrows). The looping connections imply that a node feeds its own output value back to itself on the next timestep. Each connection in the diagram has an evolvable weight: none are hard-wired by the user.

5.1 ANN Parameters to Evolve

Your EA will need to evolve the following parameters:

1. Weights for each connection in the CTRNN, including those on arcs out of the bias node.
2. Gains, g_i, for each of the hidden and output neurons.
3. Time constants, τ_i, for each of the hidden and output neurons.

The following ranges of these parameters (all real numbers) are known to work for the topology of Figure 3:

1. Weights (emanating from all nodes except the bias node): [-5.0, +5.0]
2. Biases (i.e., weights on the arcs emanating from the bias node): [-10.0 , 0.0]
3. Gains: [+1.0 , +5.0]
4. Time constants: [+1.0, +2.0]

Binary genes of 8 bits per parameter suffice to evolve CTRNNs that can solve the Tracker problem, using a population size of 50-100 and 100-200 generations.

5.2 Other important parameters and functions

Of course, you will need to devise a fitness function that rewards the desired behaviors: catching of small objects and avoidance of larger ones. Whether or not you give a penalty for doing the wrong thing is something with which you must experiment.

You will need to include a fair number of fitness trials for each agent. You should test each CTRNN with a random assortment of 40 falling objects, with each varying in size and horizontal start location. The vertical start location can be the same for each test, at, for example, level 15. You can also assume that the vertical velocity is one level per timestep, so a level-15 start will give the agent 15 timesteps to maneuver. The task becomes much harder if the horizontal velocity can vary among the falling objects, so it is best to use the same value for all objects. However, whether that value is 0, +1 cell per timestep, or -1 cell per timestep should not matter much: the evolving CTRNN described below should be able to find a good controller, although the straight drop is a bit easier for the agent to deal with.

To invoke both catching and avoidance behavior, the choice of training examples can be crucial. It is all too easy for evolution to focus solely on capturing or avoiding. Those individuals that do one or the other will beat more passive agents and will quickly dominate the population, but it can be hard to evolve more fine-tuned behavior later, especially if the whole population becomes dominated by catchers, or avoiders.

There are many approaches to this problem of premature convergence to a partially-good solution. These include various niching and crowding strategies, along with incremental evolution. However, one simpler strategy involves a carefully chosen set of training cases. If the width of falling objects is randomly chosen from a uniform distribution between (and including) 1 and 6, then, on average, 2 of 3 objects should be caught and the other 1 of 3 avoided. This 2 to 1 ratio of catchable to avoidable objects seems to give evolution enough incentive to evolve the proper behaviors. It will often evolve catching behavior first and then add on the avoidance after awhile. This will certainly not occur on every evolutionary run, but it should happen fairly often within the course of a few hundred generations.

6 Deliverables

1. An overview description of your system in text and diagrams. Include a description of your genotype representation and all parameters used for both the evolving CTRNNs and the Tracker scenarios. **3 points**

2. Verify that your system can evolve tracker agents to properly catch ALL objects (even the large ones). Include sufficient statistics (over at least 20 runs) to convince the reader that such behavior can be easily evolved using the CTRNN described above. Clearly document any parameter settings that seemed to be critical to achieving success. Give a complete formulation of the fitness function for this task. **2 points**

3. Verify that your system can evolve a single tracker agent that both catches the small and avoids the large objects. Include sufficient statistics to convince the reader that such behavior does arise occasionally, i.e. is not a once-in-your-lifetime rarity. It need not be perfect behavior, but when you watch the graphic display, you should see good signs of both catching and avoidance, though the agent may make occasional mistakes. Give a complete formulation of the fitness function that you used, along with any important aspects of the training trials. Include at least one plot of fitness as a function of the evolutionary generation, and in one paragraph, explain the basic behavior of the agent (as you see it on the graphic display) for one particular successful run. For example, does the agent wait under objects for a period, sizing them up, before making a move; or does it move continuously and only stop when it senses a small object; or does it do something else? **4 points**.

4. Make at least one significant modification to the tracker scenario (such as falling velocity of objects, maximum speed of agent, width or height of the arena, etc.) and explore how this affects the success of tracker evolution. Clearly document the modification, then present and explain the results. **2 points**

5. Make at least one significant modification to the CTRNN topology - such as changing the number of nodes in the hidden layer or some aspect of connection scheme - and explore how this affects the success of tracker evolution. Clearly document the modification, then present and explain the results. **2 points**

6. Make at least one significant modification to the range of one of the evolvable CTRNN variables (e.g. weights, biases, gains or time constants) and explore how this affects the results. Clearly document the change, then present and explain the results. **2 points**

7. In significant detail, analyze an evolved CTRNN that successfully performs the complete tracker task. If you were not able to evolve such a Tracker, then use one that performs the catch-all-objects task. To get any credit for this, you must look at both the weights of the network and the general input/output behavior of the network. For the latter, you will need to isolate the network and feed it with particular input patterns, such as all 1's, indicating that all shadow sensors are active, some 1's on the left, some 1's on the right, etc. The outputs for these (and other) input scenarios should give some indication of how the CTRNN achieves high fitness. For weight analysis, you must compare the weighted inputs to different neurons to try to find some patterns that would support potentially useful input/output behaviors. In short, you need to give a convincing explanation of agent function in terms of the CTRNN details. **5 points**

6.1 Important Details

- **Your report must not exceed SIX pages in length. Longer reports can result in a loss of points. Be clear and concise in your writing, but provide each and every piece of information requested above.**
- You may work alone, or in groups of total size 2 or 3, but no larger.
- Each GROUP should write ONE report, but all GROUP MEMBERS must upload that same report to It's Learning under their individual names.
- ALL group members must attend the demonstration session.

6.2 General Comments

There are many different network topologies for Randall Beer's CTRNNs. Nearly all of his papers involve a slightly different topology and/or range of parameter settings. You are free to use any of them, but those presented above are known to work on versions of the Tracker problem described above.

If for any reason you find that a different network topology and/or set of evolved-parameter ranges works better with your EA, then feel free to use that ANN topology and evolutionary regime as your basic framework for questions 1-3 above. Then modify it for questions 5-6.

References

[1] R. Beer, *On the dynamics of small continuous-time recurrent neural networks*, Adaptive Behavior, 3 (1995), pp. 469–509.

[2] ——, *Toward the evolution of dynamical neural networks for minimally cognitive behavior*, in From Animals to Animats 4: Proc. 4th Int. Conf. on Simulation of Adaptive Behavior, P. M. et. al., ed., MIT Press, 1996, pp. 421–429.

[3] ——, *The dynamics of active categorical perception in an evolved model agent*, Adaptive Behavior, 11 (2003), pp. 209–243.

[4] R. Beer and J. Gallagher, *Evolving dynamical neural networks for adaptive behavior*, Adaptive Behavior, 1 (1992), pp. 91–122.